哈尔滨工业大学(深圳)

《编译原理》实验报告

学　院: 计算机科学与技术

姓　名: 金正达

学　号: 220110515

专　业: 计算机科学与技术

日　期: 2024-11-12

1 实验目的与方法

实验目的为本次实验的实验目的，方法为所使用的语言，软件环境等。

1.1 词法分析器

实验目的：

1. 加深对词法分析程序的功能及实现方法的理解。

2. 对类 C 语言单词符号的文法描述有更深的认识，理解有限自动机、编码表和符号表在编译的整个过程中的应用。

3. 设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词法分析，加深对高级语言的认识。

实验方法：

Java 21.0.2 LST

Windows 11

IntelliJ IDEA

1.2 语法分析

实验目的：

1.加深对词法分析程序的功能及实现方法的理解；

2. 对类 C 语言的文法描述有更深的认识，理解有穷自动机、编码表和符号表在 编译的整个过程中的应用；

3. 设计并编程实现一个词法分析程序，对类 C 语言源程序段进行词

法分析，加深对高级语言的认识。

实验方法：

Java 21.0.2 LST

Windows 11

IntelliJ IDEA

1.3 典型语句的语义分析及中间代码生成

实验目的：

1. 采用实验二中的文法，为语法正确的单词串设计翻译方案，完成语法制导翻 译。

2. 利用该翻译方案，对所给程序段进行分析，输出生成的中间代码序列和更新后 的符号表，并保存在相应文件中。

3. 实现声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。

4. 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性。

实验方法：

Java 21.0.2 LST

Windows 11

IntelliJ IDEA

1.4 目标代码生成

实验目的：

1. 加深编译器总体结构的理解与掌握；
2. 掌握常见的 RISC-V 指令的使用方法；
3. 理解并掌握目标代码生成算法和寄存器选择算法。

实验方法：

Java 21.0.2 LST

Windows 11

IntelliJ IDEA

2 实验内容及要求

每次实验室的实验内容和要求描述清楚。

2.1 词法分析器

编写一个词法分析程序，读取文件，对文件内的类 C 语言程序段进行词法分析。

输入：以文件形式存放的类 C 语言程序段；

输出：以文件形式存放的 TOKEN 串和简单符号表。

2.2 语法分析

1. 利用 LR(1)分析法，设计语法分析程序，对输入单词符号串进行语法分析，输出推导过程中所用产生式序列并保存在输出文件中；

2. 文法支持变量申明、变量赋值、基本算术运算；实验一的输出作为实验二的输入。

2.3 典型语句的语义分析及中间代码生成

1. 采用实验二中的文法，为语法正确的单词串设计翻译方案，完成语法制导翻译。

2. 利用翻译方案，对所给程序段进行分析，输出生成的中间代码序列和更新后的符号表，并保存在相应文件中。

3. 实现声明语句、简单赋值语句、算术表达式的语义分析与中间代码生成。

4. 使用框架中的模拟器 IREmulator 验证生成的中间代码的正确性

2.4 目标代码生成

1. 将实验三生成的中间代码转换为目标代码（RISC-V 指令）；

2. 运行生成的目标代码，验证结果的正确性。

3 实验总体流程与函数功能描述

3.1词法分析

3.1.1 编码表

int 1

return 2

= 3

, 4

Semicolon 5

+ 6

- 7

* 8

/ 9

（ 10

） 11

id 51 内部字符串

IntConst 52 整数值

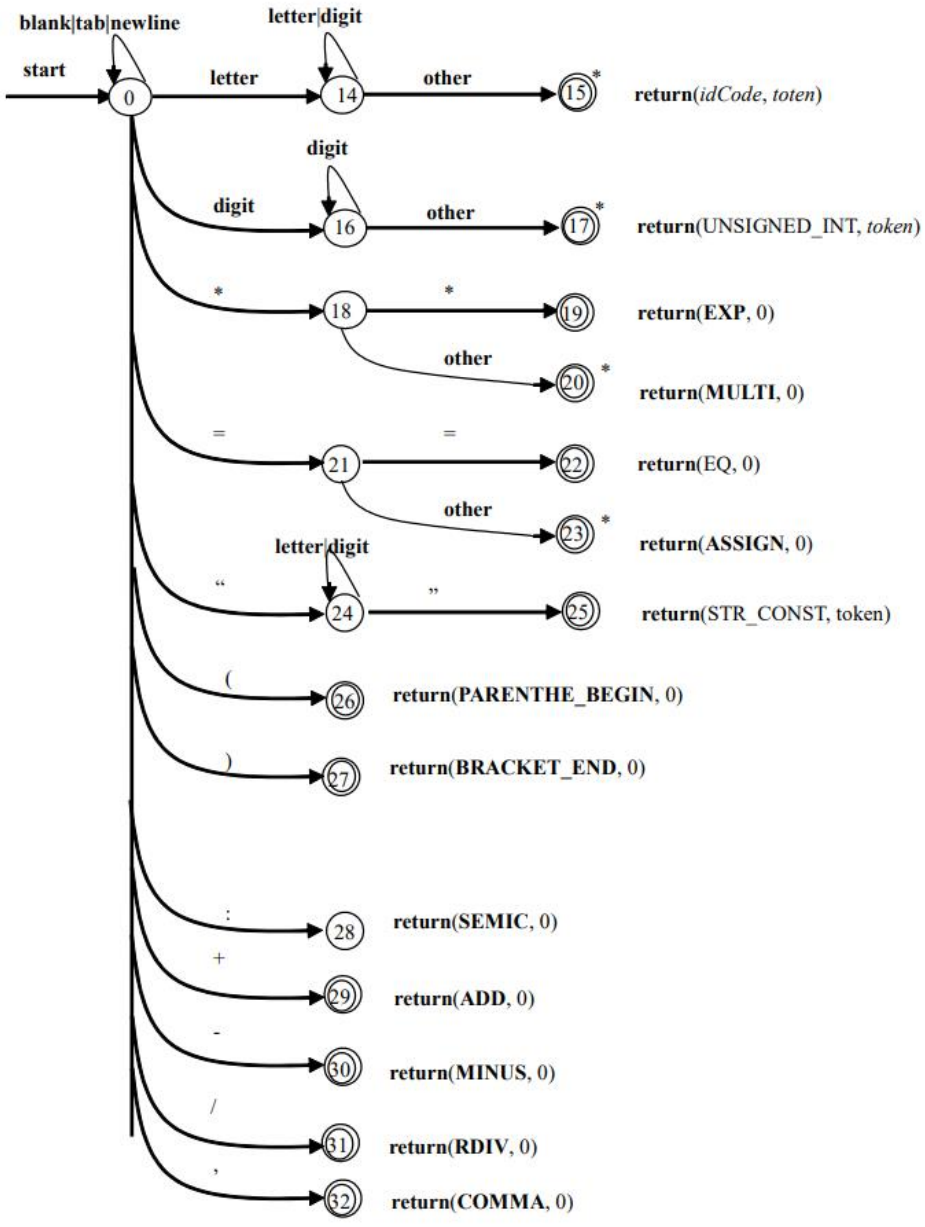
3.1.2 正则文法

标识符：id→letter (letter|digit)*

整常数：id→no_0_digit(digit|)*

符号：id → (|)|+|-|*|/|=/,;

3.1.3 状态转换图



3.1.4 词法分析程序设计思路和算法描述

设计思路：

该词法分析器的设计基于有限状态机的概念。

1.状态定义：

定义了两个主要状态：INITIAL（初始状态）和 IN_TOKEN（正在读取 token）。

2.状态转换：

状态转换基于输入字符的类型（空白字符、分号、其他字符）。

3. Token 生成：

使用 StringBuilder 来累积字符，直到遇到 token 的边界（空白字符或分号）。

4.特殊字符处理：

分号被视为特殊字符，单独生成一个 token。

算法描述：

1.初始化：

创建一个 StringBuilder 用于累积字符。

将当前状态设置为 INITIAL。

2.主循环：

对源代码的每一行的每个字符进行遍历：

a. 如果当前状态是 INITIAL：

- 如果字符是空格，保持 INITIAL 状态。
- 如果字符是分号，生成分号 token，保持 INITIAL 状态。
- 否则，将字符添加到 StringBuilder，转换到 IN_TOKEN 状态。

b. 如果当前状态是 IN_TOKEN：

- 如果字符是空格，生成当前累积的 token，清空 StringBuilder，转换到 INITIAL 状态。

- 如果字符是分号，生成当前累积的 token，清空 StringBuilder，生成分号 token，转换到 INITIAL 状态。

- 否则，将字符添加到 StringBuilder，保持 IN_TOKEN 状态。

3.后处理：

遍历结束后，如果 StringBuilder 不为空，生成最后一个 token。

生成结束标记"$"。

tokenGenerator 函数会根据输入的字符串将其转化为对应的 Token。

3.2 语法分析

3.2.1 拓展文法

P -> S_list

S_list -> S Semicolon S_list

S_list -> S Semicolon

S -> D id

D -> int

S -> id = E

S -> return E

E -> E + A

E -> E - A

E -> A

A -> A * B

A -> B

B -> (E)

B -> id

B -> IntConst

3.2.2 LR1 分析表

LR1分析表

状态	ACTION												GOTO				
	id	(	)	+	-	*	=	int	return	IntConst	Semicolon	$	E	S_list	S	A	B	D
0	shift 4							shift 5	shift 6					1		2		3
1												accept						
2											shift 7							
3	shift 8																	
4							shift 9											
5	reduce D -> int																	
6	shift 13	shift 14								shift 15			10			11	12	
7	shift 4							shift 5	shift 6			reduce S_list -> S Semicolon		16		2		3
8											reduce S -> D id							
9	shift 13	shift 14								shift 15			17			11	12	
10				shift 18	shift 19						reduce S -> return E							
11				reduce E -> A	reduce E -> A	shift 20					reduce E -> A							
12				reduce A -> B	reduce A -> B	reduce A -> B					reduce A -> B							
13				reduce B -> id	reduce B -> id	reduce B -> id					reduce B -> id							
14	shift 24	shift 25								shift 26			21			22	23	
15				reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst					reduce B -> IntConst							
16												reduce S_list -> S Semicolon S_list						
17				shift 18	shift 19						reduce S -> id = E							
18	shift 13	shift 14								shift 15						27	12	
19	shift 13	shift 14								shift 15						28	12	
20	shift 13	shift 14								shift 15							29	
21			shift 30	shift 31	shift 32													
22			reduce E -> A	reduce E -> A	reduce E -> A	shift 33												
23			reduce A -> B	reduce A -> B	reduce A -> B	reduce A -> B												
24			reduce B -> id	reduce B -> id	reduce B -> id	reduce B -> id												
25	shift 24	shift 25								shift 26			34			22	23	
26			reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst	reduce B -> IntConst												
27				reduce E -> E + A	reduce E -> E + A	shift 20					reduce E -> E + A							
28				reduce E -> E - A	reduce E -> E - A	shift 20					reduce E -> E - A							
29				reduce A -> A * B	reduce A -> A * B	reduce A -> A * B					reduce A -> A * B							
30				reduce B -> (E)	reduce B -> (E)	reduce B -> (E)					reduce B -> (E)							
31	shift 24	shift 25								shift 26						35	23	
32	shift 24	shift 25								shift 26						36	23	
33	shift 24	shift 25								shift 26							37	
34			shift 38	shift 31	shift 32													
35			reduce E -> E + A	reduce E -> E + A	reduce E -> E + A	shift 33												
36			reduce E -> E - A	reduce E -> E - A	reduce E -> E - A	shift 33												
37			reduce A -> A * B	reduce A -> A * B	reduce A -> A * B	reduce A -> A * B												
38			reduce B -> (E)	reduce B -> (E)	reduce B -> (E)	reduce B -> (E)												

3.2.3 状态栈和符号栈的数据结构和设计思路

状态栈使用栈数据结构（Stack<Status>）来跟踪当前分析的状态，

这种设计允许后进先出的特性，适合于语法分析中状态的管理。

符号栈（Stack<Symbol>）用于存储已处理的符号，帮助追踪当前输入符号的产生式。

两个栈协同工作，确保在遇到不同的语法分析动作时（如移入、规约），能够有效管理状态和符号。

3.2.4 LR 驱动程序设计思路和算法描述

LR 驱动程序的核心是通过读取输入符号并根据当前状态和输入符号决定采取的动作（移入、规约或接受）。

算法首先将初始状态压入状态栈，然后遍历输入符号列表。

在每一步，根据当前状态和输入符号查询 LR 分析表以获取动作。

如果是移入，则将新的状态和符号推入各自的栈；

如果是规约，则根据产生式弹出符号，并将新的非终结符压入符号栈，再根据新符号更新状态；

如果是接受，程序结束。错误处理则通过输出错误信息进行。这个过程反复进行，直到输入被完全处理。

3.3 语义分析和中间代码生成

3.3.1 翻译方案

```
P -> S_list

S_list -> S Semicolon S_list

S_list -> S Semicolon
```

```
S -> D id{ id.type = D.type; }
D -> int{ D.type = int; }
S -> id = E{ gencode(id.val = E.val); }
S -> return E{ gencode(return E.val); }
E -> E1 + A { E.val = E1.val + A.val; }
E -> E1 - A { E.val = E1.val - A.val; }
E -> A { E.val = A.val; }
A -> A1 * B { A.val = A1.val * B.val; }
A -> B { A.val = B.val; }
B -> ( E ) { B.val = E.val; }
B -> id { B.val = id.val; }
B -> IntConst { B.val = IntConst.lexval; }
```

3.3.2 语义分析和中间代码生成的数据结构

语义分析采用一个符号栈来存储分析过程中的终结符和非终结符，push 和 pop 操作与语法分析同步。

中间代码生成采用一个符号栈和一个指令列表，分别存储终结符和非终结符，生成的中间指令。

3.3.3 语法分析程序设计思路和算法描述

设计思路：

符号表管理：通过符号表来管理程序中的标识符及其相关信息。在分析过程中，程序会根据不同的产生式更新符号表中的标识符类型。

符号栈：使用一个栈来跟踪当前解析的符号和非终结符。通过栈的后进先出特性，可以方便地管理上下文信息。

动作观察者：实现了 ActionObserver 接口，定义了在不同解析动作（接受、规约、移入）时的具体操作。

算法描述

当接受时 (whenAccept)：

无操作。

当规约时 (whenReduce)：

根据产生式的索引来决定规约的操作：

S -> D id：弹出栈顶的 id 和 D。如果 id 是有效的（即不为 null），则更新符号表中对应 id 的类型为 D 的类型，并将新创建的非终结符推入栈中。

D -> int：弹出栈顶的 D，将其类型设置为 int，并推入新的非终结符。

对于其他产生式：按规约的规则弹出相应数量的符号，并将新非终结符推入栈中。

当移入时 (whenShift)：

创建一个新的 Symbol 实例并将当前的 Token 赋值给它。如果当前 Token 的类型是 int，则设置该符号的类型为 Int。将这个符号推入符号栈。

3.4 目标代码生成

3.4.1 设计思路和算法描述

加载前端代码：

遍历 originInstructions 中的每条 Instruction，通过 instruction.getKind() 检查其类型。

对于 ADD 类型指令：

获取左操作数 lhs 和右操作数 rhs。如果 lhs 是立即数且 rhs 是中间代码变量，则将操作数重新排序，创建一个新的 ADD 指令，将立即数放在右操作数的位置，以便于代码生成。

对于 SUB 类型指令：

同样获取 lhs 和 rhs。如果 lhs 是立即数且 rhs 是 IR 变量，为了方便处理立即数，生成一条 MOV 指令，将 lhs 移动到一个临时变量 IRVariable.temp()。创建一个新的 SUB 指令，将临时变量作为左操作数，原右操作数 rhs 不变。

最后，将处理过的指令添加到 instructions 列表中。

代码生成：

寄存器分配算法负责给变量分配寄存器，具体过程如下：

若变量已有映射寄存器，则直接返回。

否则，查找空闲寄存器，将其分配给该变量并更新映射表。

若无空闲寄存器，则优先查找占用的临时寄存器，重新分配给变量并更新映射。

执行代码生成的流程：

遍历 instructions 中的每条指令，基于指令类型生成目标代码。

对于 MOV 指令，若源操作数是立即数，则生成加载指令 li；若是变量，则生成数据传送指令 mv。

对于 ADD 指令，若右操作数是立即数，则生成加法立即数指令 addi；若是变量，则生成标准加法指令 add。

对于 SUB 和 MUL 指令，分别生成 sub 和 mul 指令。

对于 RET 指令，将返回值存入 a0 寄存器，表示函数返回值。

4 实验结果与分析

对实验的输入输出结果进行展示与分析。注意：要求给出编译器各阶段（词法分析、语法分析、中间代码生成、目标代码生成）的输入输出并进行分析说明。

词法分析：
输入：类 C 语言文件

```
int result;
int a;
int b;
int c;
a = 8;
b = 5;
c = 3 - a;
result = a * b - ( 3 + b ) * ( c - a );
return result;
```

输出：Tokens 文件，符号表。
Tokens 文件存储了输入代码中的 Tokens（token 类型和 token 内容）：

```
(int,)
(id,result)
(Semicolon,)
(int,)
(id,a)
(Semicolon,)
(int,)
(id,b)
(Semicolon,)
(int,)
(id,c)
(Semicolon,)
(id,a)
(- )
```

符号表存储了输入代码中的符号与其类型信息：

```
(a, null)
(b, null)
(c, null)
(result, null)
```

语法分析：

输入：LR（1）分析表，符号表，Tokens 文件

LR（1）分析表由外部软件根据文法生成：

```
状态,ACTION,,,,,,,,,,,,GOTO,,,,,
,id,(,),+,-,*,=,int,return,IntConst,Semicolon,$,E,S_list,S,A,B,D
0,shift 4,,,,,,,shift 5,shift 6,,,,,1,2,,,3
1,,,,,,,,,,,,accept,,,,,,
2,,,,,,,,,,,shift 7,,,,,,,
3,shift 8,,,,,,,,,,,,,,,,,
4,,,,,,,shift 9,,,,,,,,,,,
5,reduce D -> int,,,,,,,,,,,,,,,,,
6,shift 13,shift 14,,,,,,,,shift 15,,,10,,,11,12,
7,shift 4,,,,,,,shift 5,shift 6,,,reduce S_list -> S Semicolon,,16,2,,,3
8,,,,,,,,,,,reduce S -> D id,,,,,,,
9,shift 13,shift 14,,,,,,,,shift 15,,,17,,,11,12,
10,,,,shift 18,shift 19,,,,,,reduce S -> return E,,,,,,,
11,,,,reduce E -> A,reduce E -> A,shift 20,,,,,reduce E -> A,,,,,,,
12,,,,reduce A -> B,reduce A -> B,reduce A -> B,,,,,reduce A -> B,,,,,,,
13,,,,reduce B -> id,reduce B -> id,reduce B -> id,,,,,reduce B -> id,,,,,,,
14,shift 24,shift 25,,,,,,,,shift 26,,,21,,,22,23,
```

符号表和 Tokens 文件为词法分析过程输出。

输出：产生式序列文件

产生式序列：

```
D -> int
S -> D id
D -> int
S -> D id
D -> int
S -> D id
D -> int
S -> D id
B -> IntConst
A -> B
E -> A
S -> id = E
B -> IntConst
A -> B
E -> A
S -> id = E
B -> IntConst
A -> B
```

语义分析和中间代码生成：

输入：语法文件，LR（1）分析表

```
P -> S_list;
S_list -> S Semicolon S_list;
S_list -> S Semicolon;
S -> D id;
D -> int;
S -> id = E;
S -> return E;
E -> E + A;
E -> E - A;
E -> A;
A -> A * B;
A -> B;
B -> ( E );
B -> id;
B -> IntConst;

```

输出：中间代码，新的符号表

语义分析后获得带有正确类型的符号表：

```
(a, Int)
(b, Int)
(c, Int)
(result, Int)

```

中间代码：

```
(MOV, a, 8)
(MOV, b, 5)
(SUB, $0, 3, a)
(MOV, c, $0)
(MUL, $1, a, b)
(ADD, $2, 3, b)
(SUB, $3, c, a)
(MUL, $4, $2, $3)
(SUB, $5, $1, $4)
(MOV, result, $5)
(RET, , result)
```

目标代码生成：

输入：语法分析和语义分析所得的中间代码

输出：汇编代码：

```
.text
    li t4, 8
    li t5, 5
    li t6, 3
    sub t0, t6, t4
    mv t1, t0
    mul t2, t4, t5
    addi t3, t5, 3
    sub t6, t1, t4
    mul t6, t3, t6
    sub t6, t2, t6
    mv t6, t6
    mv a0, t6

```

5 实验中遇到的困难与解决办法

描述实验中遇到的困难与解决办法，对实验的意见与建议或收获。

中间代码生成需要深入理解语法树的结构和语义信息传递，要逐步对数据结构的操作进行分析，希望实验指导书对这部分的指导可以更详细一点。

语法分析的难度并不是很大，四个学时应该足够了，可以安排一个简单的代码优化实验，使实验对编译过程的覆盖更全面。